Table of Contents

1 Introduction .............................................. 1
   1.1 Installing the Interface .............................. 1

2 sm_int (library) ........................................... 2
   2.1 Usage and interface (sm_int) ........................... 3
   2.2 Documentation on exports (sm_int) .................... 3
      smcInit / 0 (pred)
      smcAddRule / 2 (pred)
      smcRetractRule / 2 (pred)
      smcCommitProgram / 0 (pred)
      smComputeModel / 0 (pred)
      smExamineModel / 2 (pred)
      smEnd / 0 (pred)

3 xnmr (library) ............................................ 36
   3.1 The XNMR Interpreter ................................. 6
   3.2 The nmr mode ........................................... 6
      3.2.1 Using the s operator ............................... 7
      3.2.2 Using the t operator ............................... 8
   3.3 Usage and interface (xnmr) ............................ 9
   3.4 Documentation on exports (xnmr) ...................... 9
      set_answerset_mode / 0 (pred)
      - / 1 (pred)
      wfs / 0 (pred)

4.1 Usage and interface (xnmr_int) .......................... 10
4.2 Documentation on exports (xnmr_int) .................... 10
      pstable_model / 3 (pred)
      atom_handle / 2 (pred)
      in_current_stable_model / 1 (pred)
      current_stable_model / 1 (pred)
      in_all_stable_models / 2 (pred)

References .................................................. 412

Predicate Definition Index ................................. 13

Global Index ................................................ 14
1 Introduction

The XASP interface provides various levels of interfaces to Smodels (V. 2.26) API. At the lowest level a raw interface allows a programmer who has translated atoms of a stable model into integer values to use the Smodels API directly.

However, most users who want to directly program with Smodels, will likely use the cooked interface, which allows arbitrary weight constraint rules, and in which atoms and literals are translated to Smodels format in a transparent manner. Using this interface, efficient XSB-based grounders can be written for special purpose programming tasks.

The above interfaces allow an XSB programmer to specify a program whose stable model is to be obtained by explicitly adding rules to a cache. However it is often useful to implicitly set up such a program, sending the residual program constructed by XSB during query evaluation to Smodels. This level of interaction is supported by the interface to the xnmr module which also provides a specially-written command-line interpreter.

Examples of use of the various interfaces can be found in the subdirectory intf_examples

1.1 Installing the Interface

In order to use the Smodels interface, several steps must be performed. Note: this interface has not yet been ported to Windows Cygwin.

1. Smodels itself must be compiled as a library. In version 2.26, this is done by typing

   make lib

   in the Smodels2.26 directory (NOT this directory). The compilation is complete if a .libs subdirectory was created containing various archive files.

2. XSB must be configured with the 'with-smodels=〈path-to-smodels〉' option. This will create a file 'smoMakefile' in this directory with a variable SMODELS set to the proper path. If this doesn’t work on your system, the SMODELS variable in the smoMakefile may need to be edited by hand.

3. The Smodels files must be made available to this interface. One way of doing this is by executing the command

   sh makelinks.sh 〈path-to-smodels〉

4. Finally,

   make -f smoMakefile
This interface allows efficient access from XSB to the C-level Smodels API. When using the interface, Smodels can be linked to XSB so that it can be used to evaluate combinatorial or satisfiability problems that may be difficult or inefficient to do directly in XSB.

This interface contains two levels: the *cooked* level and the *raw* level. The cooked level intern rules in an XSB clause store, and translates general weight constraint rules [BibRef: SiNS02] into a *normal form* that the Smodels engine can evaluate. When the programmer has determined that enough clauses have been added to the store to form a semantically complete sub-program, the program is *committed*. This means that information in the clauses is copied to Smodels and interned using Smodels data structures so that stable models of the clauses can be computed and examined. By convention, the cooked interface ensures that the atom *true* is present in all stable models, and the atom *false* is false in all stable models. The raw level models closely the Smodels API, and demands, among other things, that each atom in a stable sub-program has been translated into a unique integer. The raw level also does not provide translation of arbitrary weight constraint rules into the normal form required by the SModels engine. As a result, the raw level is significantly more difficult to directly use than the cooked level. While we make public the APIs for both the raw and cooked level, we provide support only for users of the cooked interface.

As mentioned above Smodels extends normal programs to allow weight constraints, which can be useful for combinatorial problems. However, the syntax used by Smodels for weight constraints does not follow ISO Prolog syntax so that the XSB syntax for weight constraints differs in some respects from that of SM-stats. Our syntax is defined as follows, where $A$ is a Prolog atom, $N$ a non-negative integer, and $I$ an arbitrary integer.

- **GeneralLiteral ::=** WeightConstraint $|$ Literal
- **WeightConstraint ::=** weightConst(Bound,WeightList,Bound)
- **WeightList ::=** "List of WeightLiterals"
- **WeightLiteral ::=** Literal $|$ weight(Literal,N)
- **Literal ::=** $A$ $|$ not($A$)
- **Bound ::=** $I$ $|$ undef

Thus an example of a weight constraint might be:

```
weightConst(1,[weight(a,1),weight(not(b),1)],2)
```

We note that if a user does not wish to put an upper or lower bound on a weight constraint, she may simply set the bound to `undef` or to an integer less than 0.

The intuitive semantics of a weight constraint `weightConst(Lower,WeightList,Upper)`, in which `List` is is list of `WeightLiterals` that it is true in a model $M$ whenever the sum of the weights of the literals in the constraint that are true in $M$ is between the lower `Lower` and `Upper`. Any literal in a `WeightList` that does not have a weight explicitly attached to it is taken to have a weight of 1.

In a typical session, a user will initialize the Smodels interface, add rules to the clause store until it contains a semantically meaningful sub-problem. He can then specify a compute statement if needed, commit the rules, and compute and examine stable models via
backtracking. If desired, the user can then re-initialize the interface, and add rules to or retract rules from the clause store until another semantically meaningful sub-program is defined; and then commit, compute and examine another stable model\footnote{Currently, only normal rules can be retracted.}.

Neither the raw nor the cooked interface currently supports explicit negation.

The process of adding information to a store and periodically evaluating it is vaguely reminiscent of the Constraint Logic Programming (CLP) paradigm, but there are important differences. In CLP, constraints are part of the object language of a Prolog program: constraints are added to or projected out of a constraint store upon forward execution, removed upon backwards execution, and iteratively checked. When using this interface, on the other hand, an XSB program essentially acts as a compiler for the clause store, which is treated as a target language. Clauses must be explicitly added or removed from the store, and stable model computation cannot occur incrementally – it must wait until all clauses have been added to the store. We note in passing that the \textsc{xnmr} module provides an elegant but specialized alternative. \textsc{xnmr} integrates stable models into the object language of XSB, by computing "relevant" stable models from the the residual answers produced by query evaluation. It does not however, support the weighted constraint rules, compute statements and so on that this module supports.

2.1 Usage and interface (\texttt{sm\_int})

- **Exports:**
  - **Predicates:**
    - \texttt{smcInit/0}, \texttt{smcReInit/0}, \texttt{smcAddRule/2}, \texttt{smcRetractRule/2}, \texttt{smcSetCompute/1}, \texttt{smcCommitProgram/0}, \texttt{smComputeModel/0}, \texttt{smcExamineModel/1}, \texttt{smcExamineModel/2}, \texttt{smEnd/0}, \texttt{print\_cache/0}.
- **Other modules used:**
  - **System library modules:**
    - \texttt{basics, machine}.

2.2 Documentation on exports (\texttt{sm\_int})

\texttt{smcInit/0}:

[PREDICATE]

Initializes the XSB clause store and the Smodels API. This predicate must be executed before building up a clause store for the first time. The corresponding raw predicate, \texttt{smrInit(Num)}, initializes the Smodels API assuming that it will require at most \texttt{Num} atoms.
smcReInit/0: [PREDICATE]
Reinitializes the Smodels API, but does not affect the XSB clause store. This predicate
is provided so that a user can reuse rules in a clause store in the context of more
than one sub-program.

smcAddRule/2: [PREDICATE]
smcAddRule(Head,Body) interns a ground rule into the XSB clause store. Head must
be a GeneralLiteral as defined at the beginning of this section, and Body must be
a list of GeneralLiterals. Upon interning, the rule is translated into a normal form,
if necessary, and atoms are translated to unique integers. The corresponding raw
predicates, smrAddBasicRule/3, smrAddChoiceRule/3, smrAddConstraintRule/4,
and smrAddWeightRule/3 can be used to add raw predicates immediately into the
SModels API.

smcRetractRule/2: [PREDICATE]
smcRetractBasicRule(Head,Body) retracts a ground (basic) rule from the XSB
clause store. Currently, this predicate cannot retract rules with weight constraints:
Head must be a Literal as defined at the beginning of this section, and Body must be
a list of GeneralLiterals.

smcSetCompute/1: [PREDICATE]
smcSetCompute(+List) requires that List be a list of literals – i.e. atoms or the
default negation of atoms). This predicate ensures that each literal in List is present
in the stable models returned by Smodels. By convention the cooked interface ensures
that true is present and false absent in all stable models. After translating a literal
it calls the raw interface predicates smrSetPosCompute/1 and smrSetNegCompute/1

smcCommitProgram/0: [PREDICATE]
This predicate translates all of the clauses from the XSB clause store into the data
structures of the Smodels API. It then signals to the API that all clauses have been
added, and initializes the Smodels computation. The corresponding raw predicate,
smrCommitProgram, performs only the last two of these features.

smComputeModel/0: [PREDICATE]
This predicate calls Smodels to compute a stable model, and succeeds if a stable model
can be computed. Upon backtracking, the predicate will continue to succeed until all
stable models for a given program cache have been computed. smComputeModel/0 is
used by both the raw and the cooked levels.
smcExamineModel/1:  [PREDICATE]
smcExamineModel/(-Atoms) returns the list of atoms that are true in the most recently computed stable model. To examine the truth values of particular literals, use smcExamineModel/2.

smcExamineModel/2:  [PREDICATE]
smcExamineModel/(+List,-Atoms) filters the literals in List to determine which are true in the most recently computed stable model. These true literals are returned in the list Atoms. smrExamineModel(+N,-Atoms) provides the corresponding raw interface in which integers from 0 to N, true in the most recently computed stable model, are input and output.

smEnd/0:  [PREDICATE]
Reclaims all resources consumed by Smodels and the various APIs. This predicate is used by both the cooked and the raw interfaces.

print_cache/0:  [PREDICATE]
This predicate can be used to examine the XSB clause store, and may be useful for debugging.
3 xnmr (library)

The XNMR module attempts to provide an unified environment for extending the well-founded semantics provided by XSB to the stable. It consists of a new command-line interpreter for XSB which benefits from the integration of XSB with the stable model generator Smodels [BibRef: NiSi97] and XSB to compute stable and answer set semantics for logic programs.

3.1 The XNMR Interpreter

The XNMR interpreter is the interface between XNMR and the user. It adds extra functionalities to the standard interpreter of XSB to provide means to compute different semantics for Prolog programs.

There are two distinct modes for the XNMR interpreter. The nmr mode allows for the computation of (partial) stable model semantics and answer set semantics of normal logic programs. The answerset mode is designed to compute the non-explosive partial answer set semantics of logic programs with explicit negation.

3.2 The nmr mode

The nmr mode is the default mode of XNMR. It provides support for computing the stable model and answer set semantics of normal logic programs. This mode can be identified by the nmr| ?-. If you change the mode to a different one, you can come back to nmr mode using the predicate set.

The prompt works much like a standard Prolog top-level. The difference arises when the user issues a query which has an answer that depends on undefined goals, i.e. a non-empty delay list. In this case, the user can either ignore the delay list and assume the query (with the given instantiations for the variables) is undefined under the well-founded semantics, or it can ask XNMR to call Smodels to compute different semantics for the residual program.

Three options are available in such cases:
1. ['s'] computes and prints all (partial) stable models of the residual program, one at a time;
2. ['t'] computes and prints the (partial) stable models of the residual program where the query is true, one at a time;

Consider the program exwfs.p shown below:

```prolog
:- table win/1, p/0, q/0, r/0.

win(X) :- move(X,Y), tnot(win(Y)).

move(a,b).
move(b,a).
move(b,c).
```

1 Computation of the Answer-Set semantics is not yet supported.
move(c, d).

p :- tnot(q).
q :- tnot(p).
r :- p.
r :- q.

3.2.1 Using the s operator

Below, we show the ‘s’ operator in action. At the “more answers” prompt, when there is a delay list, the user can type s<cr> and obtain the stable models of the residual program. Note that one model is computed and printed at each time. After a model is printed, the user can ask for the next one by typing ;<cr>, or she can give up by pressing only <cr>.

```bash
> xsb xnmr
[xsb_configuration loaded]
[sysinitrc loaded]
[packaging loaded]
[sModels loaded]

XSB Version 2.01 (Gouden Carolus) of August 20, 1999
[i686-pc-linux-gnu; mode: optimal; engine: chat; scheduling: batched]

nmr| ?- [exwfs].
[exwfs loaded]

yes

nmr| ?- win(X).

X = c ? ;

X = b
DELAY LIST = [tnot(win(a))] ? s

Stable Models:
{win(a)} ? ;

{win(b)} ? ;
no

X = a
DELAY LIST = [tnot(win(b))] ? s

Stable Models:
{win(b)} ? ;

{win(a)} ? ;
no
```
3.2.2 Using the t operator

We now turn to the use of the ‘t’ option on exwfs. Note that the models computed with this option form a subset of the models computed with the ‘s’ option, namely those models where the query is true.

```
lfmobile:~/Programs/Prolog/WFS> xsb xnmr
[xsb_configuration loaded]
[sysinitrc loaded]
[packaging loaded]
[sModels loaded]

XSB Version 2.01 (Gouden Carolus) of August 20, 1999
[i686-pc-linux-gnu; mode: optimal; engine: chat; scheduling: batched]

nmr| ?- [exwfs].
[exwfs loaded]

yes
nmr| ?- win(X).

X = c ? ;

X = b
DELAY LIST = [tt(not(win(a)))] ? t

Stable Models:
{win(b)} ? ;
no

X = a
DELAY LIST = [tt(not(win(b)))] ? t

Stable Models:
{win(a)} ? ;
no

no
nmr| ?-
```
3.3 Usage and interface (xnmr)

- **Exports:**
  - **Predicates:**
    - `set_answerset_mode/0`, `set_nmr_mode/0`, `-/1`, `wfs/0`.
- **Other modules used:**
  - **System library modules:**
    - `assert`, `consult`, `file_io`, `gensym`, `loader`, `machine`, `num_vars`, `standard`, `tables`, `xsb_configuration`, `xsb_read`, `xsb_writ`.

3.4 Documentation on exports (xnmr)

**set_answerset_mode/0:**
No further documentation available for this predicate.

**set_nmr_mode/0:**
No further documentation available for this predicate.

**-/1:**
No further documentation available for this predicate.

**wfs/0:**
No further documentation available for this predicate.
4 xnmr_int (library)

This module provides the interface from the xnmr module to Smodels. It does not use the sm_int interface, but rather directly calls the Smodels C interface, and can be thought of as a special-purpose alternative to sm_int.

4.1 Usage and interface (xnmr_int)

- Exports:
  - Predicates:
    - pstable_model/3, print_current_stable_model/0, atom_handle/2,
      in_current_stable_model/1, init_smmodels/1, current_stable_model/1, in_all_stable_models/2, a_stable_model/0.
  - Other modules used:
    - System library modules:
      assert, basics, gensym, intern, setof, standard, subsumes, tables, xsb_writ.

4.2 Documentation on exports (xnmr_int)

pstable_model/3: [PREDICATE]
pstable_model(+Query,-Model,+Flag) returns nondeterministically a list of atoms true in the partial stable model total on the atoms relevant to instances of Query, if Flag is 0. If Flag is 1, it only returns models in which the instance of Query is true.

print_current_stable_model/0: [PREDICATE]
prints the current stable model to the stream to which answers are sent.(stdfbbrk)

atom_handle/2: [PREDICATE]
atom_handle(?Atom,?AtomHandle) is set by init_smmodels/1 to be an integer uniquely identifying each atoms in the residual program (and thus each atom in the Herbrand base of the program for which the stable models are to be derived. The initial query given to init_smmodels has the atom-handle of 1.

in_current_stable_model/1: [PREDICATE]
in_current_stable_model(?AtomHandle) is true of handles of atoms true in the current stable model (set by an invocation of a_stable_model/0.)
init_smodels/1:  [PREDICATE]
init_smodels(+Query): initializes smodels with the residual program produced by evaluationg Query. Query must be a call to a tabled predicate that is currently completely evaluated (and should have a delay list).

current_stable_model/1:  [PREDICATE]
current_stable_model(-AtomList) returns the list of atoms true in the current stable model.

in_all_stable_models/2:  [PREDICATE]
in_all_stable_models(+AtomHandle,+Neg) is true if Neg is 0 and the atom numbered AtomHandle is true in all stable models (of the residual program set by the previous call to init_smodels/1). If Neg is nonzero, then it is true if the atom is in NO stable model.

a_stable_model/0:  [PREDICATE]
This predicate invokes Smodels to find a (new) stable model (of the program set by the previous invocation of init_smodels/1.) It will compute all stable models through backtracking. If there are no (more) stable models, it fails. Atoms true in a stable model can be examined by in_current_stable_model/1.
References

[NS97] Ilkka Niemelä and Patrik Simons.
SModels — An implementation of the stable model and well-founded semantics for normal LP.

Extending and implementing the stable model semantics.
Predicate Definition Index

- `-` ................................. 9
  `-!/1` ............................... 9

A
  `a_stable_model/0` .............. 11
  `atom_handle/2` .................. 10

C
  `current_stable_model/1` ........ 11

I
  `in_all_stable_models/2` ........ 11
  `in_current_stable_model/1` ..... 10
  `init_smodels/1` .............. 11

P
  `print_cache/0` ................. 5

W
  `wfs/0` ........................... 9

S
  `set_answerset_mode/0` .......... 9
  `set_nnr_mode/0` ................ 9
  `smcAddRule/2` .................. 4
  `smcCommitProgram/0` .......... 4
  `smcExamineModel/1` .......... 4
  `smcExamineModel/2` .......... 5
  `smcInit/0` ...................... 3
  `smcComputeModel/0` .......... 4
  `smcReInit/0` .................. 3
  `smcRetractRule/2` .......... 4
  `smcSetCompute/1` .......... 4
  `smEnd/0` ...................... 5

P
  `print_current_stable_model/0` .... 10
  `pstable_model/3` ............ 10
# Global Index

This is a global index containing pointers to places where concepts, predicates, modes, properties, types, applications, etc., are referred to in the text of the document. Note that due to limitations of the info format unfortunately only the first reference will appear in online versions of the document.

| A          | B  | C    | D    | E    | F  | G    | H    | I    | J    | K    | L    | M    | N    | O    | P    | Q    | R    | S    | T    | U    | V    | W    | X    | Y    | Z    |
|------------|----|------|------|------|----|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|
| a_stable_model/0 | assert | current_stable_model/1 | file_io | gensym | in_all_stable_models/2 | in_current_stable_model/1 | init_smodels(+Query) | init_smodels/1 | loader | machine | num_vars | -/1 | basics | consult | print_cache/0 | print_current_stable_model/0 | pstable_model/3 | set_anserset_mode/0 | set_mnr_mode/0 | setof | smcAddRule/2 | smcCommitProgram/0 | smcExamineModel(+List,-Atoms) | smcExamineModel(-Atoms) | smcExamineModel/1 | smcExamineModel/2 | smcInit/0 | smcComputeModel/0 | smcReInit/0 | smcRetractRule/2 | smcSetCompute(+List) | smcSetCompute/1 | smEnd/0 | smrAddBasicRule/3 | smrAddChoiceRule/3 | smrAddConstraintRule/4 | smrAddWeightRule/3 | smrCommitProgram | smrExamineModel(+N,-Atoms) | smrInit(Num) | smrSetNegCompute/1 | smrSetPosCompute/1 | standard | subsumes | tables | wfs/0 | xsb_configuration | xsb_read | xsb_writ |